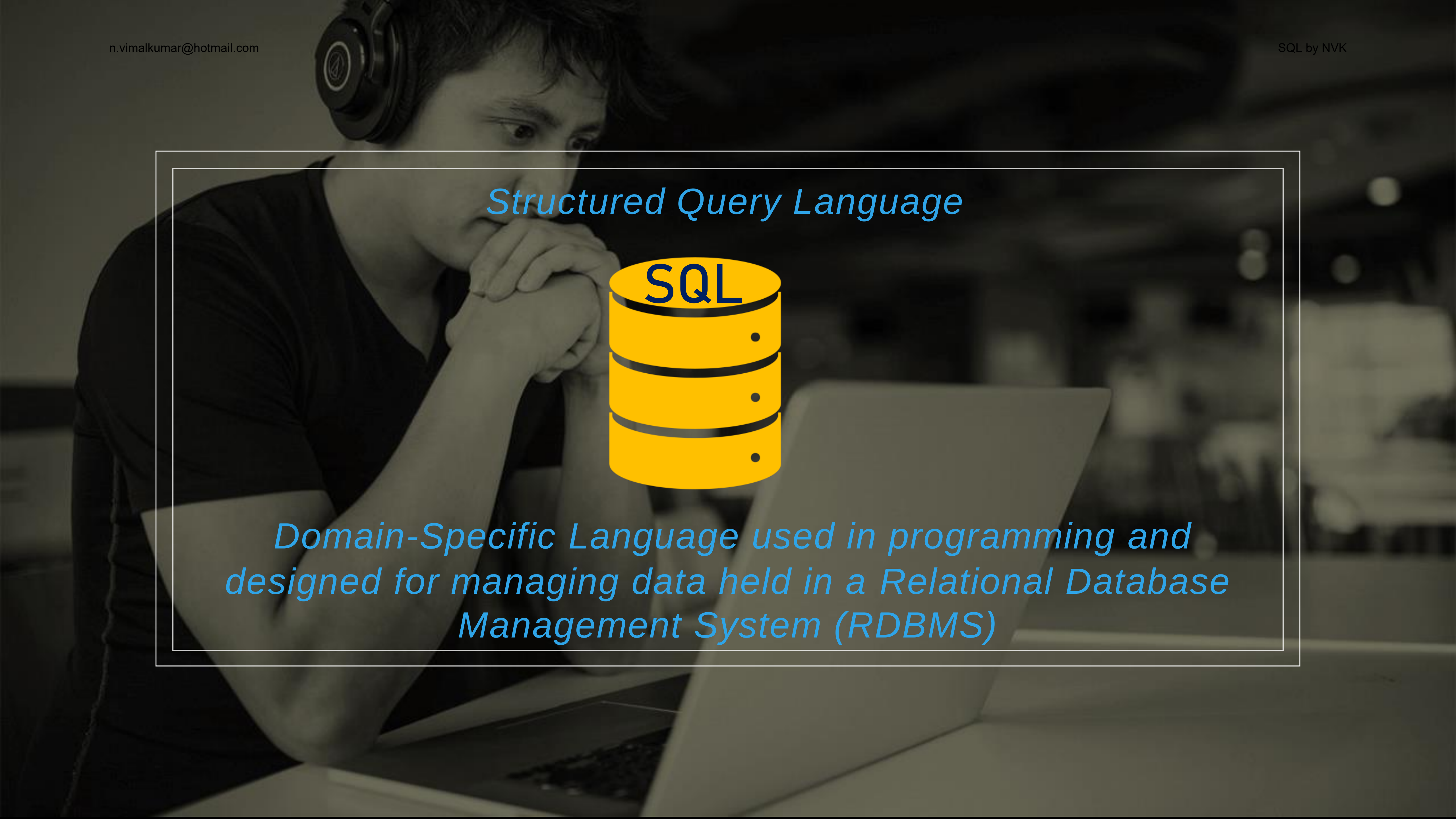
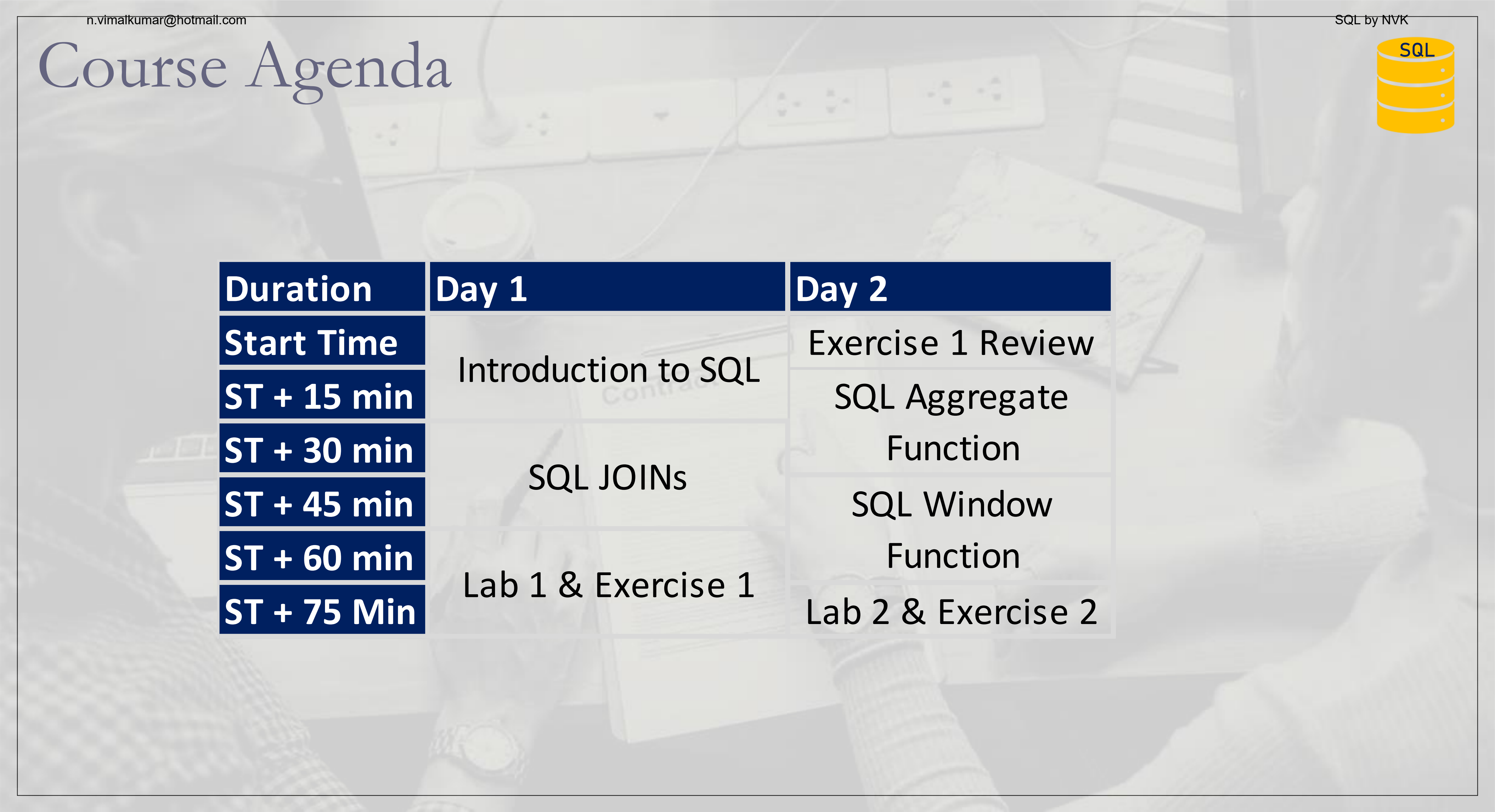


# Structured Query Language

SQL

Domain-Specific Language used in programming and designed for managing data held in a Relational Database Management System (RDBMS)

# Course Agenda



| Duration | Day 1 | Day 2 |
|---|---|---|
| Start Time | Introduction to SQL | Exercise 1 Review |
| ST + 15 min |  | SQL Aggregate Function |
| ST + 30 min | SQL JOINs |  |
| ST + 45 min |  | SQL Window Function |
| ST + 60 min | Lab 1 & Exercise 1 |  |
| ST + 75 Min |  | Lab 2 & Exercise 2 |

# Course Progress

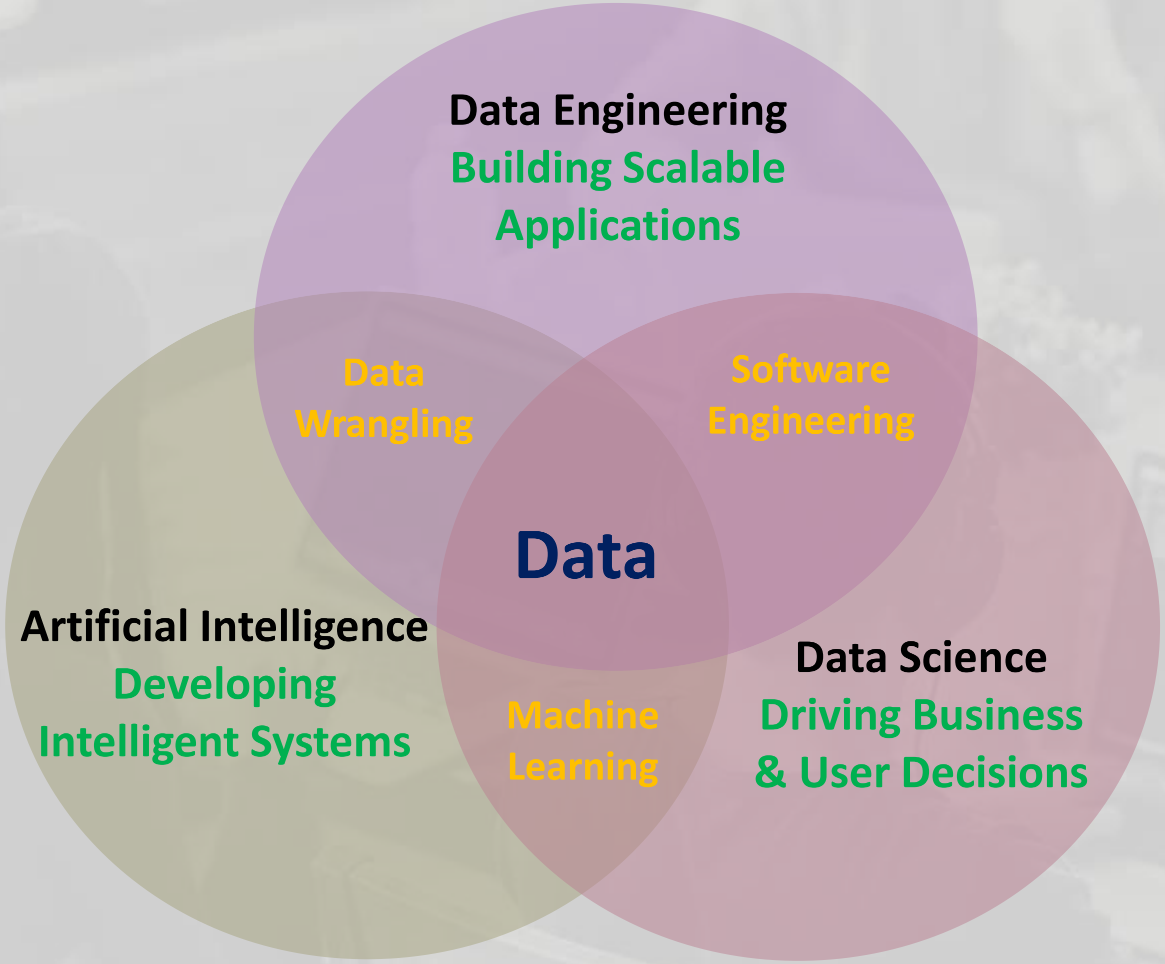
## Day 1

- Introduction to SQL
- SQL JOINs
- Lab 1 & Exercise 1

## Day 2

- Exercise 1 Review
- SQL Aggregate Function
- SQL Window Function
- Lab 2 & Exercise 2

# Vimalkumar Narasimman

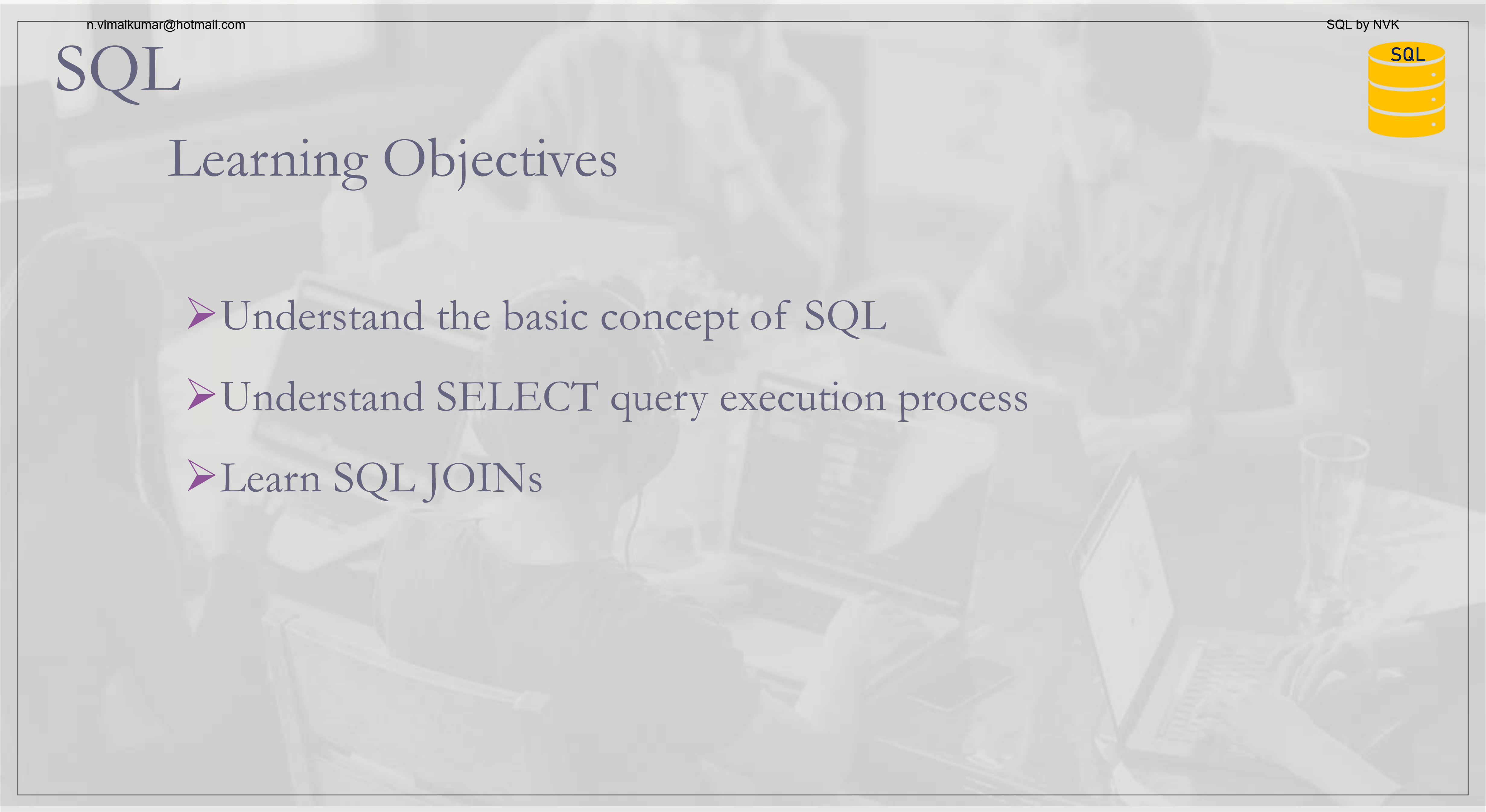


connect me on LinkedIn

skywise.

CSM CERTIFIED

THE Open GROUP TOGAF® 9 Certified



# SQL

## Learning Objectives

- Understand the basic concept of SQL
- Understand SELECT query execution process
- Learn SQL JOINs

# Basic concepts of SQL

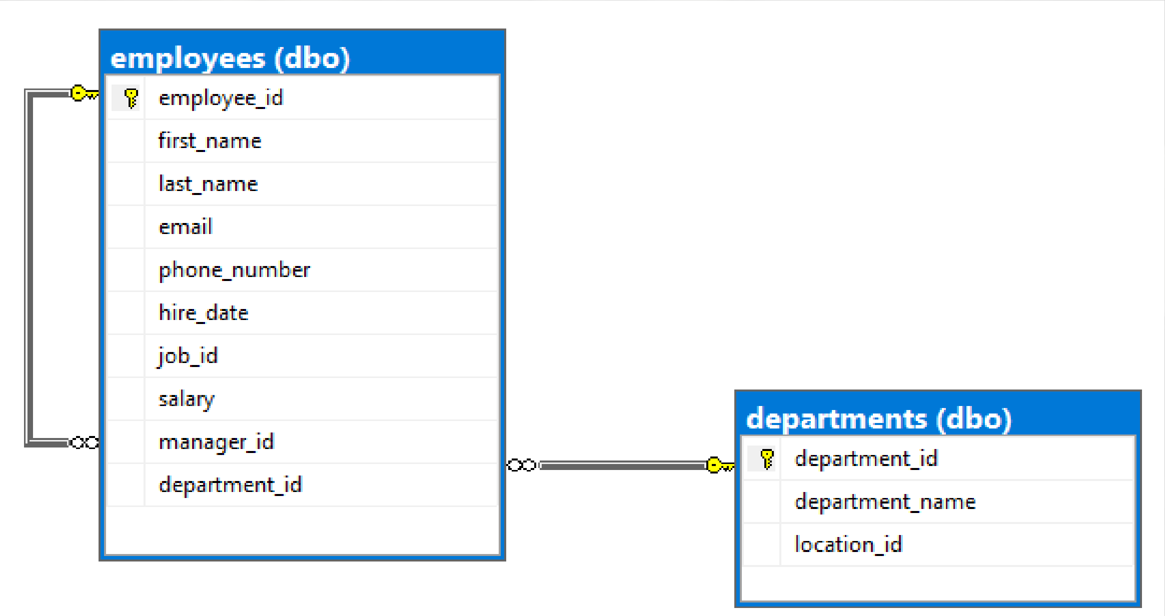
- **Data** are units of information
- Logically organized data in a row-and-column format is called **Records**
- Each row represents a unique record, and each column represents a field in the record
- A collection of related records are grouped and stored in a **Table**
- Tables are database objects that contain all the data in a **Database**

**For example**, a table that contains employee data for a company might contain a row for each employee and columns representing employee information such as employee number, name, address, job title, and home telephone number.

# Basic concepts of SQL

- A relationship is formed by correlating rows belonging to different tables
- A table relationship is established when a child table defines a Foreign Key column that references the Primary Key column of its parent table.
- **one-to-many** is the most common relationship, and it associates a row from a parent table to multiple rows in a child table.

**For example**, the “Department” and “Employee” tables have a one-to-many relationship. That is, each department have many employees. But each employee comes from only one department.



employees (dbo)

employee_id<br>first_name<br>last_name<br>email<br>phone_number<br>hire_date<br>job_id<br>salary<br>manager_id<br>department_id

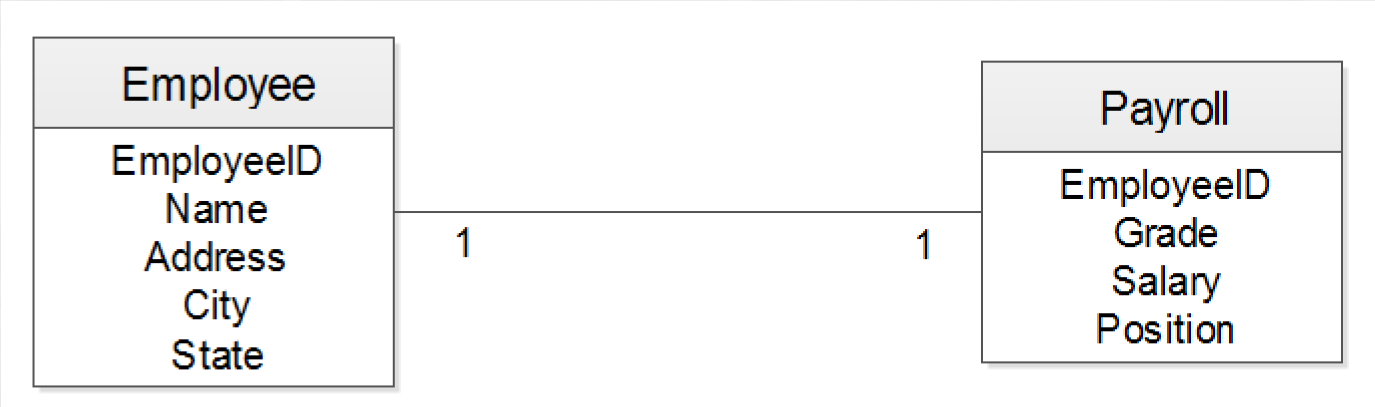
departments (dbo)

department_id<br>department_name<br>location_id

# Basic concepts of SQL

- **one-to-one** requires the child table Primary Key to be associated via a Foreign Key with the parent table Primary Key column.

**For example**, the “Employee" and “Payroll" tables have a one-to-one relationship. That is, each employee will have only one payroll.



Employee

EmployeeID
Name
Address
City
State

1 1

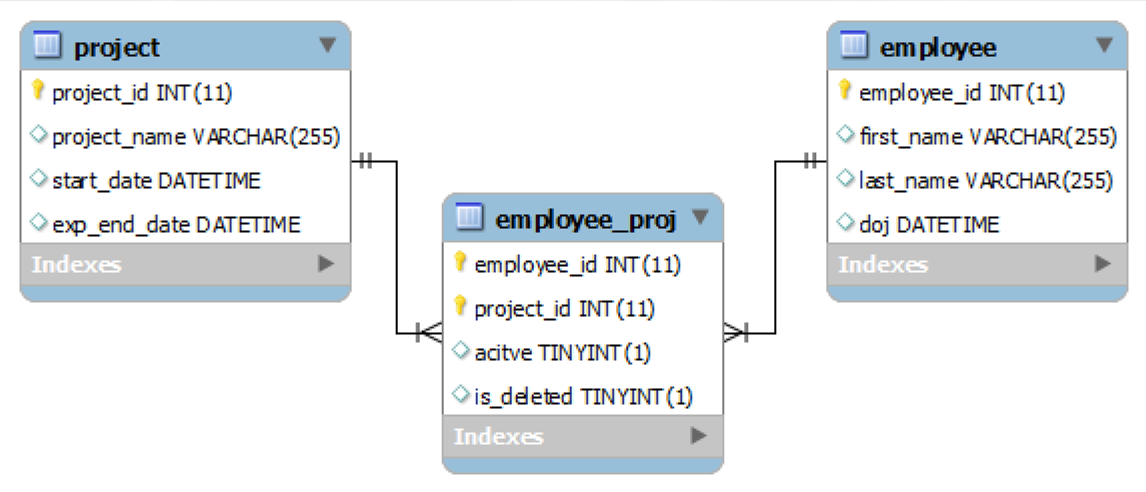
Payroll

EmployeeID
Grade
Salary
Position

# Basic concepts of SQL

❖ **many-to-many** requires a link table containing two Foreign Key columns that reference the two different parent tables.

**For example**, the “Employee" and “Project" tables have a many-to-many relationship. That is, each employee will assigned to many projects. Also each project will mapped with many employees.



project
project_id INT(11)
project_name VARCHAR(255)
start_date DATETIME
exp_end_date DATETIME
Indexes

employee_proj
employee_id INT(11)
project_id INT(11)
acitve TINYINT(1)
is_deleted TINYINT(1)
Indexes

employee
employee_id INT(11)
first_name VARCHAR(255)
last_name VARCHAR(255)
doj DATETIME
Indexes

# Basic concepts of SQL

## ❖ Transitive Dependencies

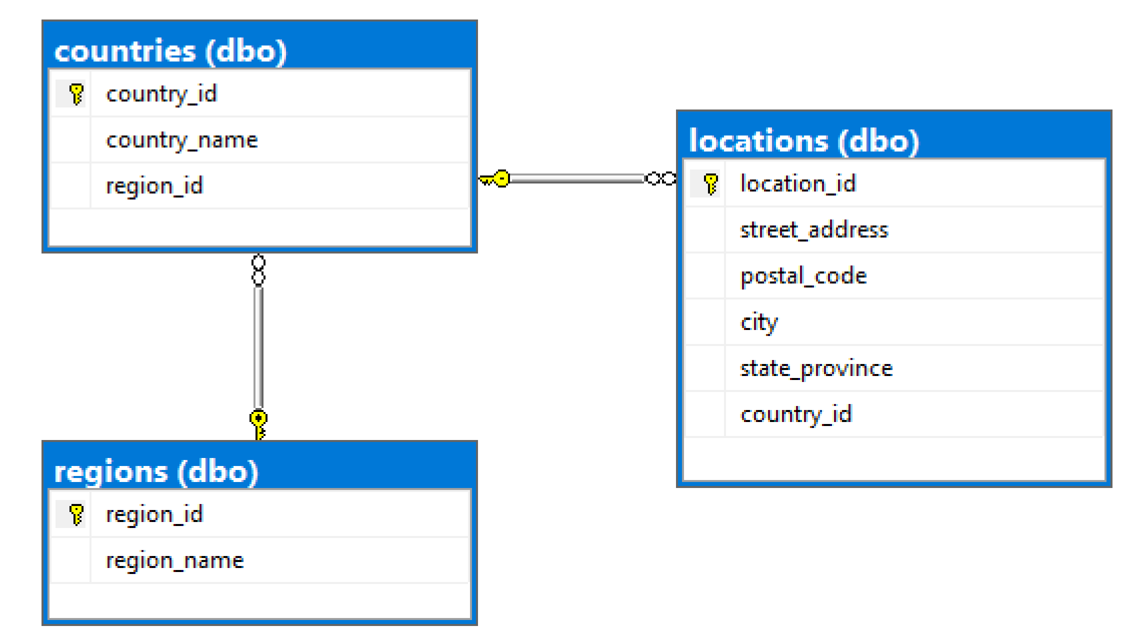
A transitive dependency exists when you have the following functional dependency pattern:

**A→B and B→C; therefore A→C**

**For example**, the “Regions“ and “Countries" tables have relationship. the “Counties“ and “Locations“ tables have relationship.

Therefore the “Regions" and “Locations" tables have functional dependence

That is, each region will have location using country records



# Simple SELECT statement

The SQL **SELECT** statement is used to retrieve records from one or more tables in your SQL database. The records retrieved are known as a result set.

The syntax for the SELECT statement in SQL is:

```
SELECT expressions
FROM tables
[WHERE conditions]
[ORDER BY expression [ ASC | DESC ]];
```

- `Expressions:` The columns or calculations that you wish to retrieve. Use * if you wish to select all columns
- `Tables:` The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause
- `WHERE conditions:` Optional. The conditions that must be met for the records to be selected
- `ORDER BY expression:` Optional. The expression used to sort the records in the result set
- `ASC:` Optional. ASC sorts the result set in ascending order by expression. This is the default behavior, if no modifier is provider
- `DESC:` Optional. DESC sorts the result set in descending order by expression

# Simple SELECT Query

| supplier_id | supplier_name | city | state |
|---|---|---|---|
| 100 | Microsoft | Redmond | Washington |
| 200 | Google | Mountain View | California |
| 300 | Oracle | Redwood City | California |
| 400 | Kimberly-Clark | Irving | Texas |
| 500 | Tyson Foods | Springdale | Arkansas |
| 600 | SC Johnson | Racine | Wisconsin |
| 700 | Dole Food Company | Westlake Village | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 900 | Electronic Arts | Redwood City | California |

```
SELECT supplier_name, city
FROM suppliers
WHERE supplier_id > 500
ORDER BY supplier_name ASC, city DESC;
```

| supplier_name | city |
|---|---|
| Dole Food Company | Westlake Village |
| Electronic Arts | Redwood City |
| Flowers Foods | Thomasville |
| SC Johnson | Racine |

This example would return only the `supplier_name` and `city` fields from the `suppliers` table where the `supplier_id` value is greater than `500`. The results are sorted by `supplier_name` in ascending order and then `city` in descending order.

# Simple SELECT statement Process Steps

1. Getting Data (**FROM** clause)
2. Row Filter (**WHERE** clause)
3. Return Expressions (**SELECT** clause)
4. Order (**ORDER BY** clause)

- The first step in the process is the execution of the statements in FROM clause
- After getting qualified rows, it is passed on to the WHERE clause. This evaluates every row using conditional expressions. When rows do not evaluate to true, they will be removed from the set
- In the next step, the processor evaluates what will be printed as a result of the query, and if there are some functions to run on data like *Distinct, Max, Sqrt, Date, Lower, etc.*
- The final processing steps of the query deal with presentation ordering the result set

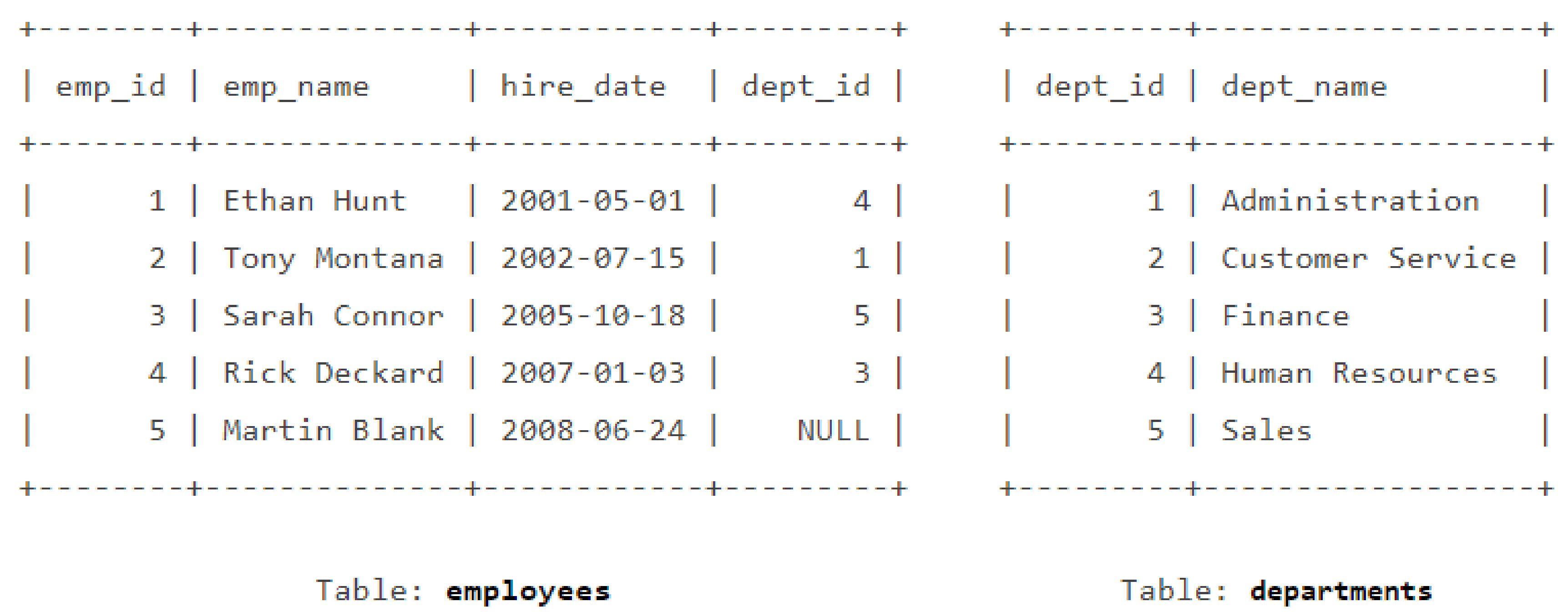
# JOINs

- A **SQL Join** statement is used to combine data or rows from two or more tables based on a common field between them.

- To understand this easily, let's look at the following employees and departments tables. Here, the dept_id column of the employees table is the foreign key to the departments table. Therefore, these two tables can be joined to get the combined data.

- Different types of Joins

  - **INNER JOIN**
  - **LEFT JOIN**
  - **RIGHT JOIN**
  - **FULL JOIN**



```
+--------+--------------+--------------+---------+     +---------+------------------+
| emp_id | emp_name     | hire_date    | dept_id |     | dept_id | dept_name        |
+--------+--------------+--------------+---------+     +---------+------------------+
|      1 | Ethan Hunt   | 2001-05-01 |       4 |     |       1 | Administration   |
|      2 | Tony Montana | 2002-07-15 |       1 |     |       2 | Customer Service |
|      3 | Sarah Connor | 2005-10-18 |       5 |     |       3 | Finance          |
|      4 | Rick Deckard | 2007-01-03 |       3 |     |       4 | Human Resources  |
|      5 | Martin Blank | 2008-06-24 |    NULL |     |       5 | Sales            |
+--------+--------------+--------------+---------+     +---------+------------------+
```

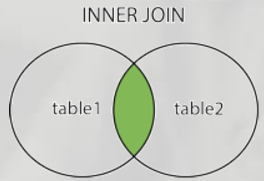
Table: **employees**

Table: **departments**

# JOINs

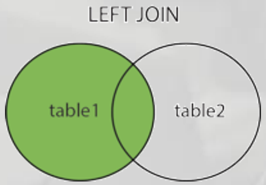
- **INNER JOIN:** Returns records that have matching values in both tables

```
SELECT emp.emp_id, emp.emp_name, emp.hire_date, dept.dept_name
FROM employees AS emp
INNER JOIN departments AS dept ON emp.dept_id = dept. dept _id
```



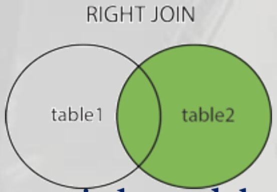
- **LEFT JOIN:** Returns all records from the left table, and the matched records from the right table

```
SELECT emp.emp_id, emp.emp_name, emp.hire_date, dept.dept_name
FROM employees AS emp
LEFT JOIN departments AS dept ON emp.dept_id = dept. dept _id
```



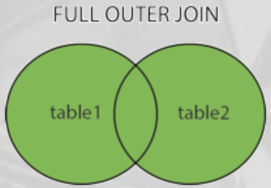
- **RIGHT JOIN:** Returns all records from the right table, and the matched records from the left table

```
SELECT emp.emp_id, emp.emp_name, emp.hire_date, dept.dept_name
FROM employees AS emp
RIGHT JOIN departments AS dept ON emp.dept_id = dept. dept _id
```



- **FULL JOIN:** Returns all records when there is a match in either left or right table

```
SELECT emp.emp_id, emp.emp_name, emp.hire_date, dept.dept_name
FROM employees AS emp
FULL OUTER JOIN departments AS dept ON emp.dept_id = dept. dept _id
```



# Exercise 1 - A

**The Table creation queries are present in this link**

This query should return only the *employee_name* , *email, phone_number, hire_date* and *salary* fields from the *employees* table where the *hire_date* value is greater than "*1990-01-01*" and *salary* in between *6,000.00* and *9999.00*. The results are sorted by *hire_date* in descending order and then *employee_name* in ascending order.

Note: Concatenate with *first_name* and *last_name* to get *employee_name*

# Exercise 1 - B

**The Table creation queries are present in this link**

This query should return employee details along with dependents details with *employee_name* , *email, phone_number, hire_date, salary, dependent_name* and *relationship* fields from the *employees* and *dependents* table where the *department_name* is “*Sales*” or “*Finance*”. The results are sorted by *employee_name* and then *dependent_name* .

Note: Concatenate with *first_name* and *last_name* to get *dependent_name*

# Course Progress

## Day 1

Introduction to SQL

SQL JOINs

Lab 1 & Exercise 1

## Day 2

Exercise 1 review

SQL Aggregate Function

SQL Window Function

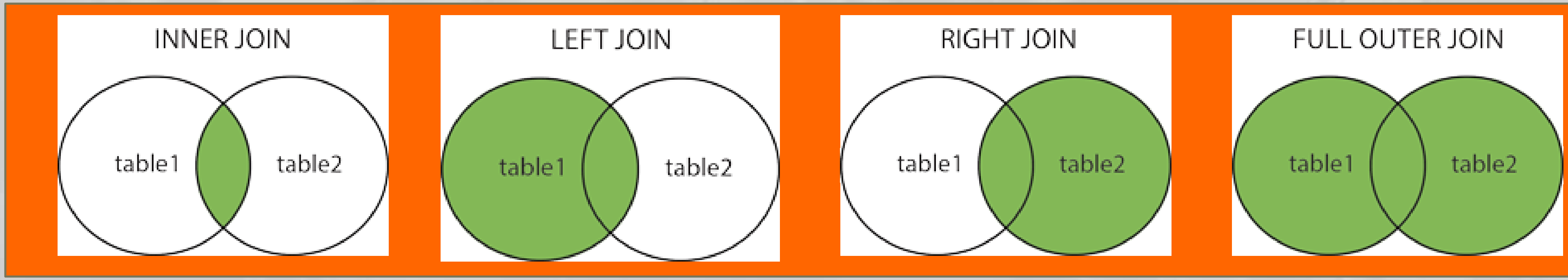
Lab 2 & Exercise 2

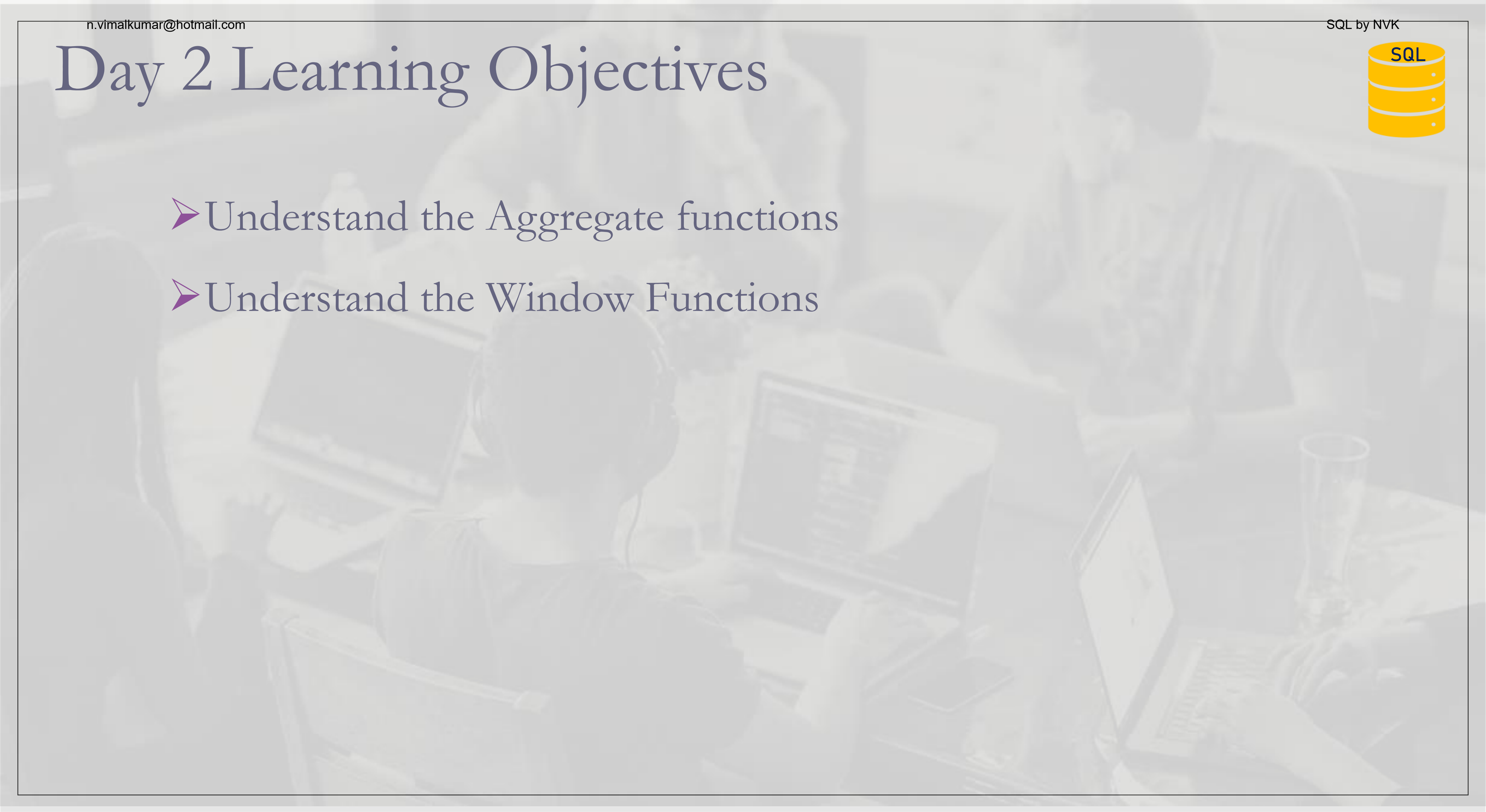
# Recap of Day 1

**Data Relationships:**

- One-to-One
- One-to-Many
- Many-to-Many

```
SELECT expressions
FROM tables
[WHERE conditions]
[ORDER BY expression [ ASC | DESC ]];
```





# Day 2 Learning Objectives

- ➢Understand the Aggregate functions
- ➢Understand the Window Functions

# Aggregate Functions

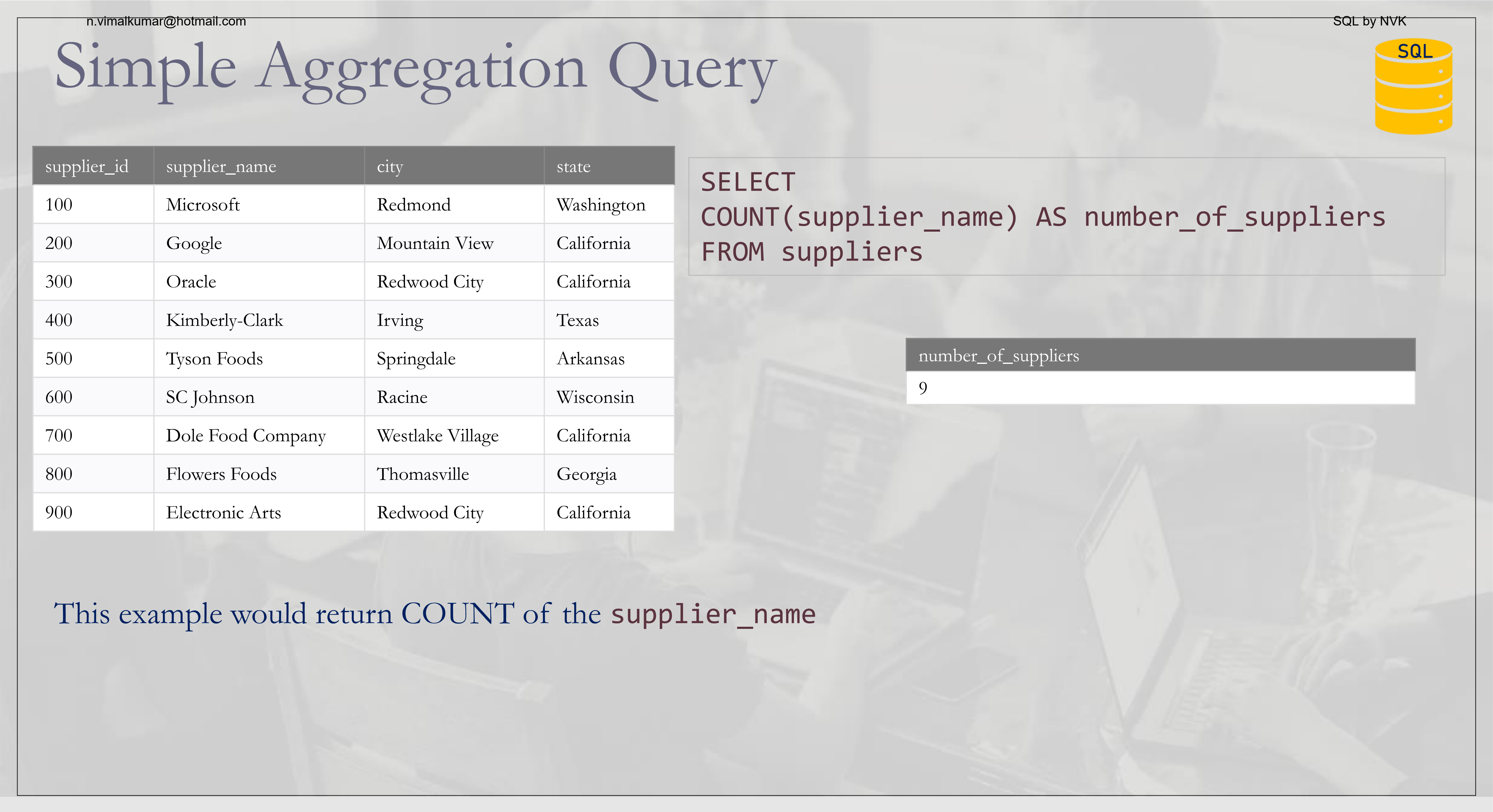
- In database management, an **aggregate function** or **aggregation function** is a function where the values of multiple rows are grouped together to form a single summary statistics

Common aggregate functions include:

- Average (i.e., arithmetic mean)
- Count
- Maximum
- Minimum
- Sum

Others Include:

- Median
- Mode
- Range
- Stddev



# Simple Aggregation Query

| supplier_id | supplier_name | city | state |
|---|---|---|---|
| 100 | Microsoft | Redmond | Washington |
| 200 | Google | Mountain View | California |
| 300 | Oracle | Redwood City | California |
| 400 | Kimberly-Clark | Irving | Texas |
| 500 | Tyson Foods | Springdale | Arkansas |
| 600 | SC Johnson | Racine | Wisconsin |
| 700 | Dole Food Company | Westlake Village | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 900 | Electronic Arts | Redwood City | California |

```
SELECT
COUNT(supplier_name) AS number_of_suppliers
FROM suppliers
```

| number_of_suppliers |
|---|
| 9 |

This example would return COUNT of the `supplier_name`

# GROUP BY and HAVING statement

The GROUP BY clause instructs the RDBMS to group the data and then perform the aggregate (function) on each group rather than on the entire result set.

Aside from the aggregate calculation statements, every column in your SELECT statement must be present in the GROUP BY clause.

The HAVING clause is used in combination with the GROUP BY clause to restrict the groups of returned rows to only those condition is TRUE

The GROUP BY clause must come after any WHERE clause and before any ORDER BY clause.

The syntax for the GROUP BY and HAVING statement in SQL is:

```
SELECT expressions
FROM tables
[WHERE conditions]
[GROUP BY expressions
        [HAVING conditions]]
[ORDER BY expression [ ASC | DESC ]];
```

# Simple Aggregation Query with GROUP BY

| supplier_id | supplier_name | city | state |
|---|---|---|---|
| 100 | Microsoft | Redmond | Washington |
| 200 | Google | Mountain View | California |
| 300 | Oracle | Redwood City | California |
| 400 | Kimberly-Clark | Irving | Texas |
| 500 | Tyson Foods | Springdale | Arkansas |
| 600 | SC Johnson | Racine | Wisconsin |
| 700 | Dole Food Company | Westlake Village | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 900 | Electronic Arts | Redwood City | California |

```
SELECT state,
COUNT(supplier_name) AS number_of_suppliers
FROM suppliers
GROUP BY state
ORDER BY state
```

| state | |
|---|---|
| Arkansas | |
| California | |
| Georgia | |
| Texas | |
| Washington | |
| Wisconsin | |

| state | number_of_suppliers |
|---|---|
| Arkansas | 1 |
| California | 4 |
| Georgia | 1 |
| Texas | 1 |
| Washington | 1 |
| Wisconsin | 1 |

# Simple Aggregation Query with GROUP BY and HAVING

| supplier_id | supplier_name | city | state |
|---|---|---|---|
| 100 | Microsoft | Redmond | Washington |
| 200 | Google | Mountain View | California |
| 300 | Oracle | Redwood City | California |
| 400 | Kimberly-Clark | Irving | Texas |
| 500 | Tyson Foods | Springdale | Arkansas |
| 600 | SC Johnson | Racine | Wisconsin |
| 700 | Dole Food Company | Westlake Village | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 900 | Electronic Arts | Redwood City | California |

```
SELECT state,
COUNT(supplier_name) AS number_of_suppliers
FROM suppliers
GROUP BY state HAVING COUNT(supplier_name) > 1
ORDER BY state
```

| state | |
|---|---|
| Arkansas | |
| California | |
| Georgia | |
| Texas | |
| Washington | |
| Wisconsin | |

| state | number_of_suppliers |
|---|---|
| Arkansas | 1 |
| California | 4 |
| Georgia | 1 |
| Texas | 1 |
| Washington | 1 |
| Wisconsin | 1 |

| state | number_of_suppliers |
|---|---|
| California | 4 |

# SELECT statement Process Steps

1. Getting Data (**FROM, JOIN** clause)
2. Row Filter (**WHERE** clause)
3. Grouping (**GROUP BY** clause)
4. Group Filter (**HAVING** clause)
5. Return Expressions (**SELECT** clause)
6. Order & Paging (**ORDER BY** clause & **LIMIT** / **OFFSET**)

# Exercise 2 - A

**The Table creation queries are present in this link**

This query should return all available *region_name, country_name* , *city, department_name* and *number_of_employee* fields from the relevant tables. The results are sorted by *country_name* and then *dependent_name* .

Note: *number_of_employee* can be zero if no employee is working in particular *region_name, country_name* , *city* **and** *dependent_name*

# Exercise 2 - B

**The Table creation queries are present in this link**

This query should return all available *region_name, country_name* , *department_name, average_employee_salary* and *recent_hiring_date* fields from the relevant tables. The results are sorted by *recent_hiring_date*.

# Window Functions

- In SQL, a **window function** or **analytic function** is a function which uses values from one or multiple rows to return a value for each row

- This contrasts with an aggregate function, which returns a single value for multiple rows

- the OVER clause defines a window or user-specified set of rows within a query result set

- A window function then computes a value for each row in the window

- You can use the OVER clause with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results

# Types of Window Functions

- Aggregate Window Functions

  `SUM(), MAX(), MIN(), AVG(), COUNT()`

- Ranking Window Functions

  `RANK(), DENSE_RANK(), ROW_NUMBER(), NTILE()`

- Analytic Window Functions

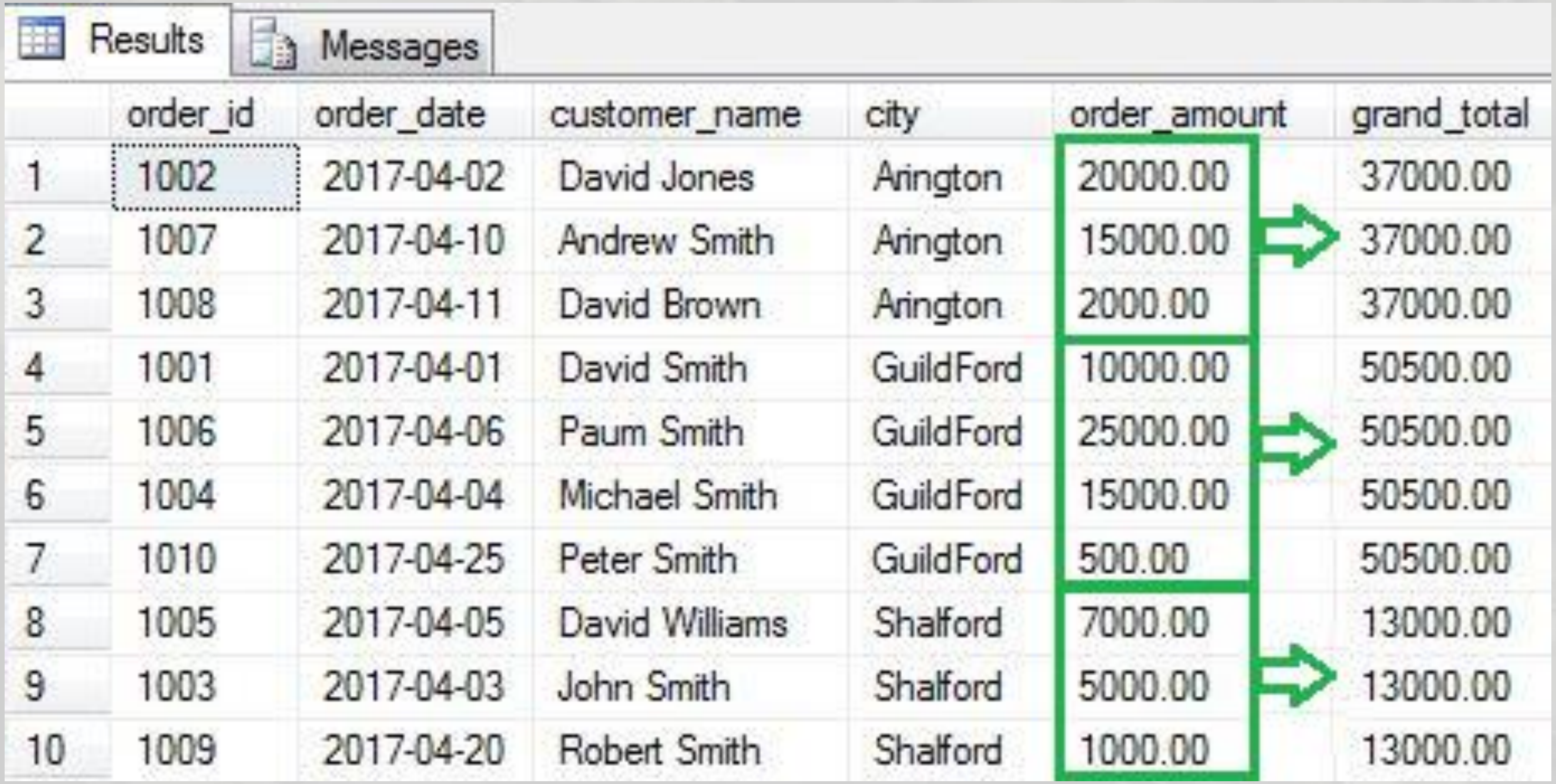
  `LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE()`

```
WINDOW_FUNCTION(expression)
OVER ( [ <PARTITION BY clause> ] [ <ORDER BY clause> ] [ <ROW or RANGE clause> ] )
```

- ✓ PARTITION BY that divides the query result set into partitions.
- ✓ ORDER BY that defines the logical order of the rows within each partition of the result set.
- ✓ ROWS/RANGE that limits the rows within the partition by specifying start and end points within the partition.

# Aggregate Window Query

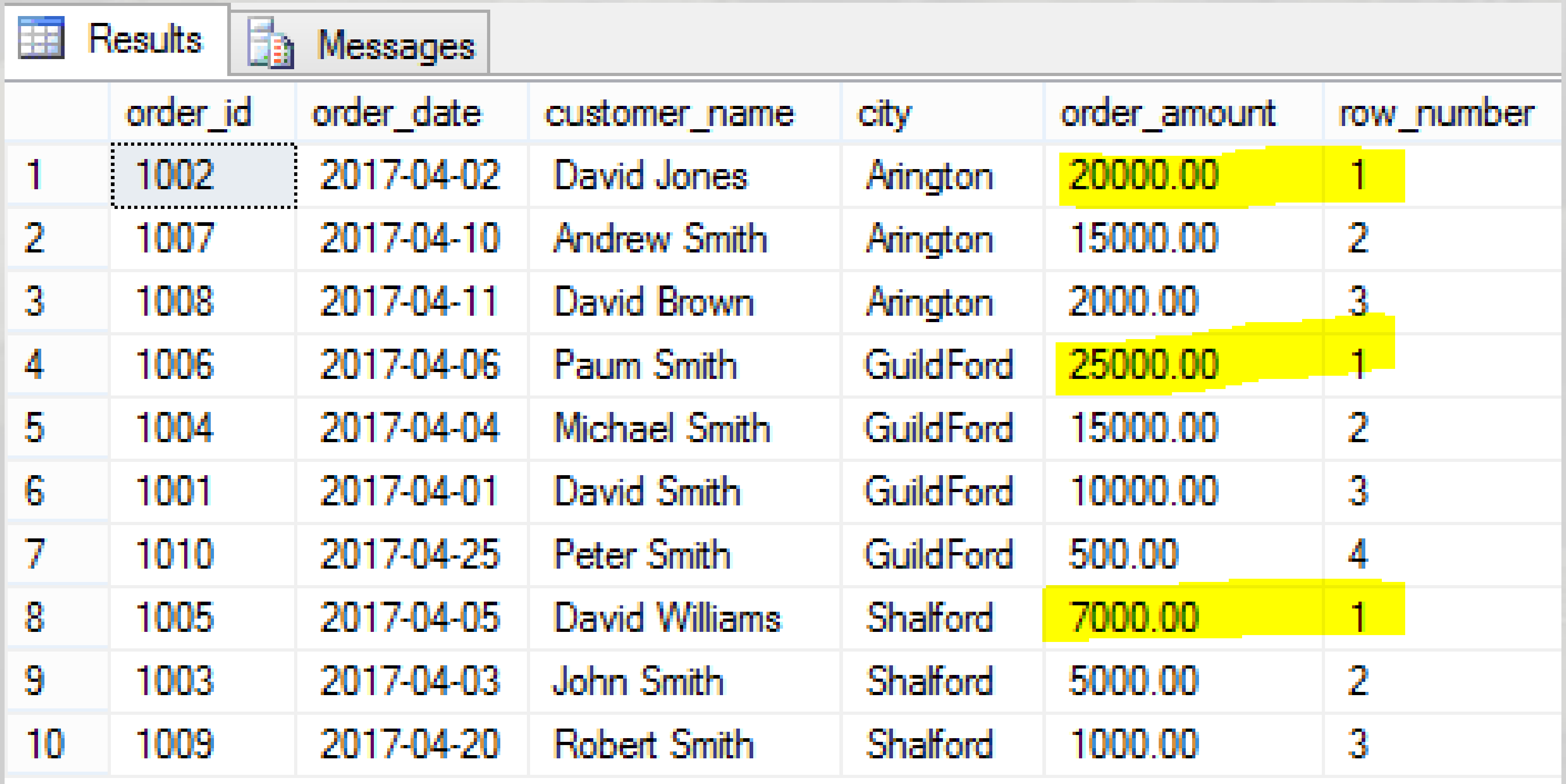
```
SELECT order_id, order_date, customer_name, city, order_amount
 ,SUM(order_amount) OVER(PARTITION BY city) as grand_total
FROM [dbo].[Orders]
```



| | order_id | order_date | customer_name | city | order_amount | grand_total |
|---|---|---|---|---|---|---|
| 1 | 1002 | 2017-04-02 | David Jones | Arington | 20000.00 | 37000.00 |
| 2 | 1007 | 2017-04-10 | Andrew Smith | Arington | 15000.00 | 37000.00 |
| 3 | 1008 | 2017-04-11 | David Brown | Arington | 2000.00 | 37000.00 |
| 4 | 1001 | 2017-04-01 | David Smith | GuildFord | 10000.00 | 50500.00 |
| 5 | 1006 | 2017-04-06 | Paum Smith | GuildFord | 25000.00 | 50500.00 |
| 6 | 1004 | 2017-04-04 | Michael Smith | GuildFord | 15000.00 | 50500.00 |
| 7 | 1010 | 2017-04-25 | Peter Smith | GuildFord | 500.00 | 50500.00 |
| 8 | 1005 | 2017-04-05 | David Williams | Shalford | 7000.00 | 13000.00 |
| 9 | 1003 | 2017-04-03 | John Smith | Shalford | 5000.00 | 13000.00 |
| 10 | 1009 | 2017-04-20 | Robert Smith | Shalford | 1000.00 | 13000.00 |

# Ranking Window Query

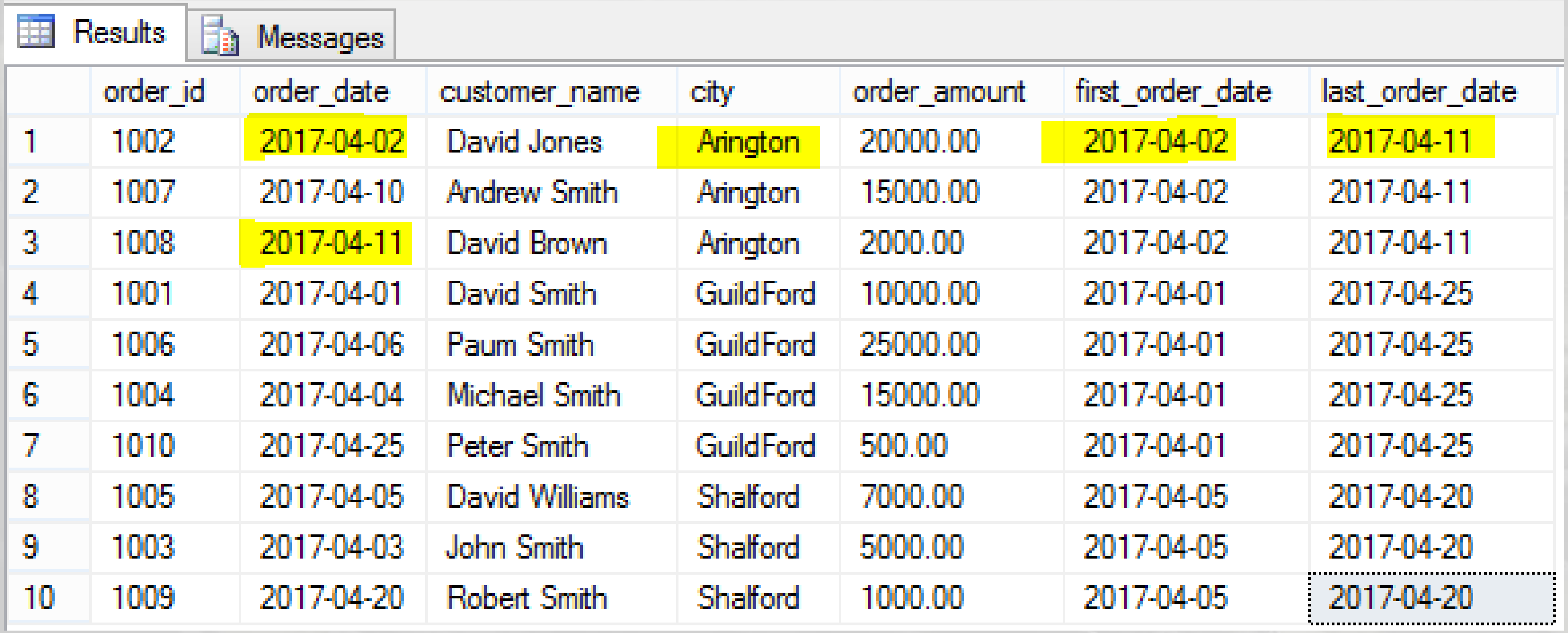
```
SELECT order_id,order_date,customer_name,city, order_amount,
ROW_NUMBER() OVER(PARTITION BY city ORDER BY order_amount DESC) [row_number]
FROM [dbo].[Orders]
```



| | order_id | order_date | customer_name | city | order_amount | row_number |
|---|---|---|---|---|---|---|
| 1 | 1002 | 2017-04-02 | David Jones | Arlington | 20000.00 | 1 |
| 2 | 1007 | 2017-04-10 | Andrew Smith | Arlington | 15000.00 | 2 |
| 3 | 1008 | 2017-04-11 | David Brown | Arlington | 2000.00 | 3 |
| 4 | 1006 | 2017-04-06 | Paum Smith | GuildFord | 25000.00 | 1 |
| 5 | 1004 | 2017-04-04 | Michael Smith | GuildFord | 15000.00 | 2 |
| 6 | 1001 | 2017-04-01 | David Smith | GuildFord | 10000.00 | 3 |
| 7 | 1010 | 2017-04-25 | Peter Smith | GuildFord | 500.00 | 4 |
| 8 | 1005 | 2017-04-05 | David Williams | Shalford | 7000.00 | 1 |
| 9 | 1003 | 2017-04-03 | John Smith | Shalford | 5000.00 | 2 |
| 10 | 1009 | 2017-04-20 | Robert Smith | Shalford | 1000.00 | 3 |

# Analytic Window Query

```
SELECT order_id,order_date,customer_name,city, order_amount,
FIRST_VALUE(order_date) OVER(PARTITION BY city ORDER BY city) first_order_date,
LAST_VALUE(order_date) OVER(PARTITION BY city ORDER BY city) last_order_date
FROM [dbo].[Orders]
```



| | order_id | order_date | customer_name | city | order_amount | first_order_date | last_order_date |
|---|---|---|---|---|---|---|---|
| 1 | 1002 | 2017-04-02 | David Jones | Arington | 20000.00 | 2017-04-02 | 2017-04-11 |
| 2 | 1007 | 2017-04-10 | Andrew Smith | Arington | 15000.00 | 2017-04-02 | 2017-04-11 |
| 3 | 1008 | 2017-04-11 | David Brown | Arington | 2000.00 | 2017-04-02 | 2017-04-11 |
| 4 | 1001 | 2017-04-01 | David Smith | GuildFord | 10000.00 | 2017-04-01 | 2017-04-25 |
| 5 | 1006 | 2017-04-06 | Paum Smith | GuildFord | 25000.00 | 2017-04-01 | 2017-04-25 |
| 6 | 1004 | 2017-04-04 | Michael Smith | GuildFord | 15000.00 | 2017-04-01 | 2017-04-25 |
| 7 | 1010 | 2017-04-25 | Peter Smith | GuildFord | 500.00 | 2017-04-01 | 2017-04-25 |
| 8 | 1005 | 2017-04-05 | David Williams | Shalford | 7000.00 | 2017-04-05 | 2017-04-20 |
| 9 | 1003 | 2017-04-03 | John Smith | Shalford | 5000.00 | 2017-04-05 | 2017-04-20 |
| 10 | 1009 | 2017-04-20 | Robert Smith | Shalford | 1000.00 | 2017-04-05 | 2017-04-20 |

# Exercise 2 - C

**The Table creation queries are present in this link**

This query should return all available *region_name, country_name* , *department_name, number_of_employee, first_joined_employee_name* and *last_joined_employee_name* fields from the relevant tables. The results are sorted by *country_name* and then *dependent_name* .

Note: *number_of_employee* can be zero, *first_joined_employee_name* and *last_joined_employee_name* can be null if no employee is working in particular *region_name, country_name* **and** *dependent_name*

# Exercise 2 - D

**The Table creation queries are present in this link**

This query should return all available *region_name, country_name* , *department_name, average_employee_salary* and *total_employee_salary* by each *region, country and department* fields from the relevant tables. The results are sorted by *recent_hiring_date*.

# Reference Link

- SQL Tutorial
- SELECT statement Process Step
- Aggregate Functions
- Window Functions

# THANK YOU!